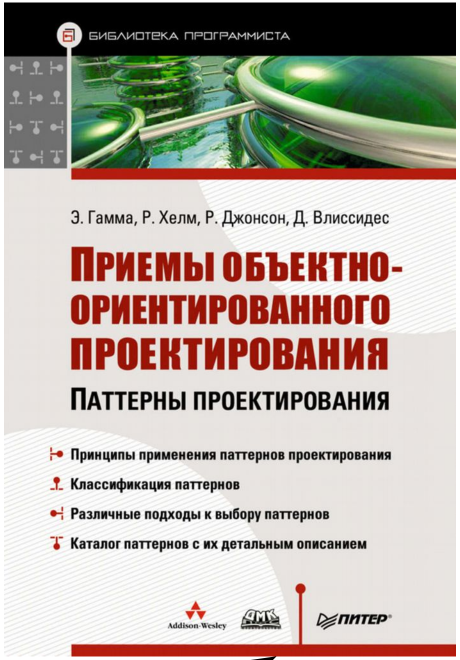
Object Oriented Design, Part 2

- Design Pattern Definition. Pattern Classification
- Common GoF Patterns
  - Strategy
  - Decorator
  - Iterator. External and Internal Iteration
  - Observer. Events and Event Objects
  - Proxy. Remote Stubs. Decorator vs Proxy
  - Facade
  - Creational Patterns: Singleton, Abstract Factory, Builder
- Some Pattern Pitfalls

Deadline-Driven Development™

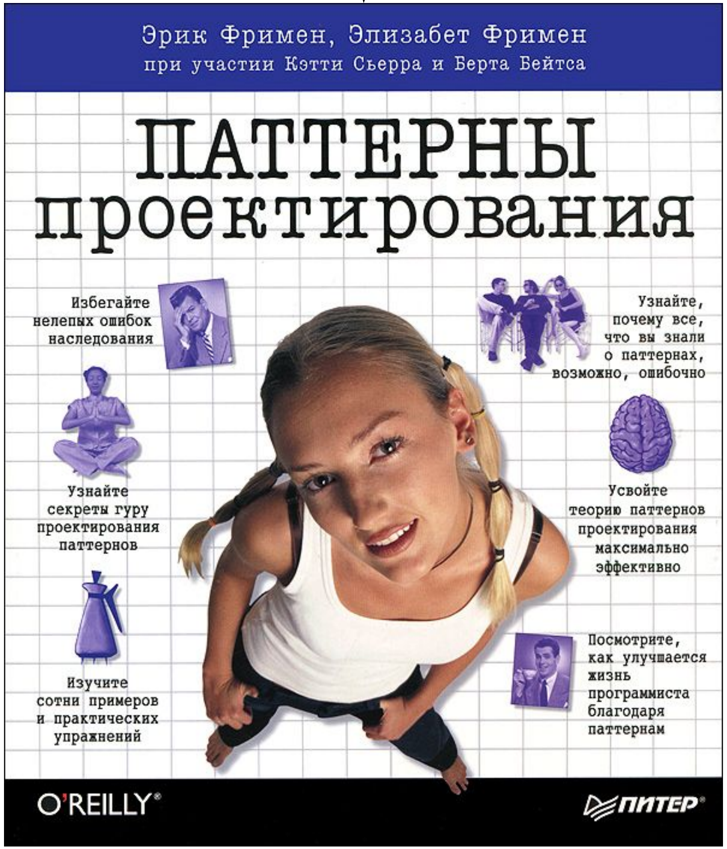
- Mar **26**: Object-Oriented Design Artifacts
  - High-Level Design, *e.g.* using CRC Cards
    - Class candidates w/public methods
    - Class Relationships and Interactions (of the form: *X* uses *Y* to accomplish *z*)
  - Detailed Design
    - System Structure, *e.g.* using UML Class Diagram
      - Classes, Public Methods. Fields ***iif*** *these are important for your design*
      - Class Relationships (Association, Aggregation, Composition) with Cardinality (1-1, 1-*, *-*)
    - System Behavior, *e.g.* using UML Sequence and/or State Machine/Activity Diagrams
      - Class Interactions
      - State Transitions
  - Ad-hoc Diagrams and Text (incl. code snippets) are **allowed**. But CRC Cards+UML are **preferred**
- **Apr 02**: First Release
  - Local Maven or Gradle build. **Must** compile and run!
  - **Should** demonstrate a basic User Story
  - Unit Tests would be good (but **not** a requirement!)


БИБЛИОТЕКА ПРОГРАММИСТА
Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес
ПРИЕМЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ
ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ
Принципы применения паттернов проектирования
Классификация паттернов
Различные подходы к выбору паттернов
Каталог паттернов с их детальным описанием
Addison-Wesley
ДМК
ПИТЕР

The Classic GoF Book

Same book, different cover

Best Intro into Patterns

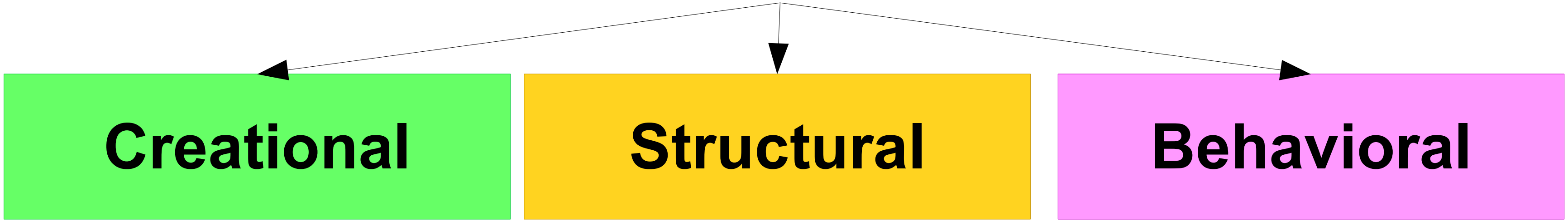

Эрик Фримен, Элизабет Фримен
при участии Кэтти Сьерра и Берта Бейтса
ПАТТЕРНЫ
проектирования
Избегайте нелепых ошибок наследования
Узнайте, почему все, что вы знали о паттернах, возможно, ошибочно
Узнайте секреты гуру проектирования паттернов
Усвойте теорию паттернов проектирования максимально эффективно
Изучите сотни примеров и практических упражнений
Посмотрите, как улучшается жизнь программиста благодаря паттернам
O'REILLY
ПИТЕР

Design Pattern is…

https://en.wikipedia.org/wiki/Software_design_pattern

- **General**, **Reusable** solution to a **commonly occurring** problem
  - within a given **Context**
  - in **Software Design**
    - *e.g.,* Christopher Alexander described patterns in **architecture**
- **Description** or **Template** that can be used in **many different situations**
  - Shows relationships and interactions between classes or objects **in general**
  - Facilitates **Common Language** between developers
- Formalized **best practice**
- Patterns are Programming Paradigm-Dependent!
  - Some OO Patterns are Functional Programming in disguise
  - *E.g.* http://www.norvig.com/design-patterns/design-patterns.pdf

Pattern Classification



Creational

- Abstract Factory
- Factory Method
- Builder
- Singleton
- Prototype
- …

Structural

- Decorator
- Proxy
- Facade
- Adapter
- Flyweight
- …

Behavioral

- Strategy
- Iterator
- Observer
- State
- Template Method
- Visitor
- …

GoF Book: Cake is a Lie!

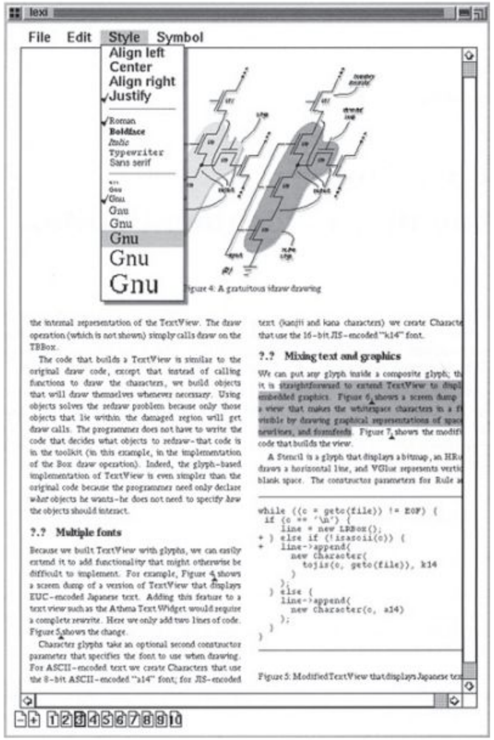


Figure 4: A gratuitous idraw drawing

the internal representation of the TextView. The draw operation (which is not shown) simply calls draw on the TBBox.

The code that builds a TextView is similar to the original draw code, except that instead of calling functions to draw the characters, we build objects that will draw themselves whenever necessary. Using objects solves the redraw problem because only those objects that lie within the damaged region will get draw calls. The programmer does not have to write the code that decides what objects to redraw–that code is in the toolkit (in this example, in the implementation of the Box draw operation). Indeed, the glyph-based implementation of TextView is even simpler than the original code because the programmer need only declare *what* objects he wants–he does not need to specify *how* the objects should interact.

?.? Multiple fonts

Because we built TextView with glyphs, we can easily extend it to add functionality that might otherwise be difficult to implement. For example, Figure 4 shows a screen dump of a version of TextView that displays EUC-encoded Japanese text. Adding this feature to a text view such as the Athena Text Widget would require a complete rewrite. Here we only add two lines of code. Figure 5 shows the change.

Character glyphs take an optional second constructor parameter that specifies the font to use when drawing. For ASCII-encoded text we create Characters that use the 8-bit ASCII-encoded "a14" font; for JIS-encoded text (kanji and kana characters) we create Characte that use the 16-bit JIS-encoded "k14" font.

?.? Mixing text and graphics

We can put any glyph inside a composite glyph; th it is straightforward to extend TextView to displ embedded graphics. Figure 6 shows a screen dump a view that makes the whitespace characters in a fi visible by drawing graphical representations of spac newlines, and formfeeds. Figure 7 shows the modifi code that builds the view.

A Stencil is a glyph that displays a bitmap, an HRu draws a horizontal line, and VGlue represents verti blank space. The constructor parameters for Rule a

```
while ((c = getc(file)) != EOF) {
 if (c == '\n') {
     line = new LRBox();
+ } else if (!isascii(c)) {
+     line->append(
        new Character(
           tojis(c, getc(file)), k14
        )
     );
   } else {
     line->append(
        new Character(c, a14)
     );
   }
}
```

Figure 5: Modified TextView that displays Japanese tex

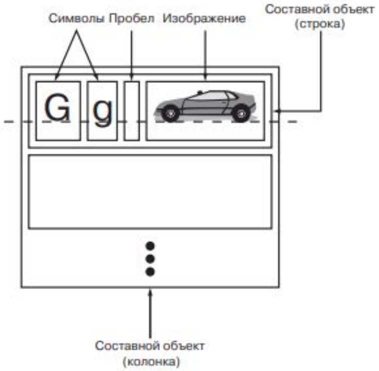
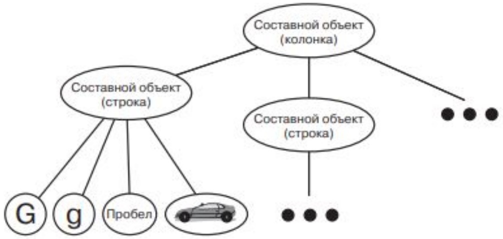


Рис. 2.2. Рекурсивная композиция текста и графики



Составной объект (колонка)
Составной объект (строка)
Составной объект (строка)
G
g
Пробел

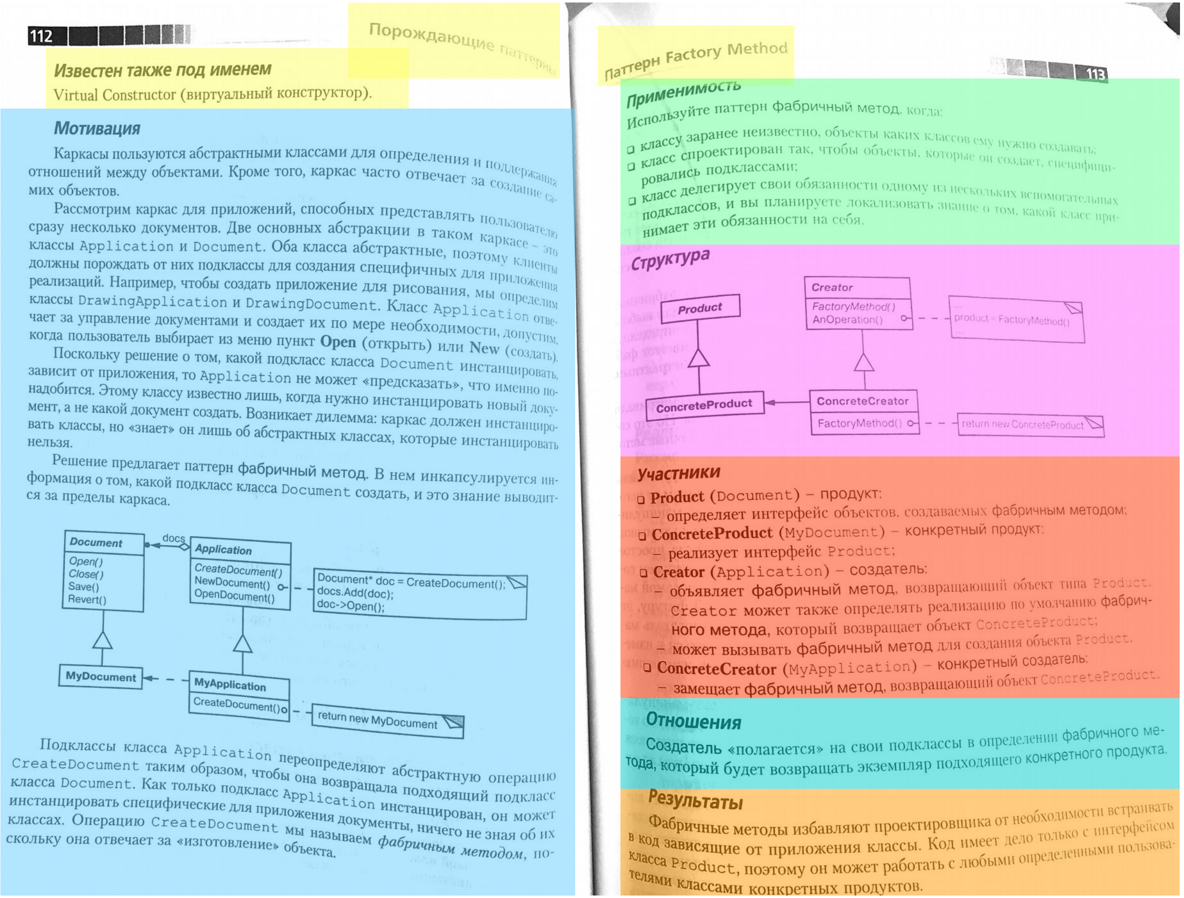
Рис. 2.3. Структура объекта для рекурсивной композиции текста и графики

The Classic *Design Patterns* book is mostly **dense text**

- Pattern Rationale
- Source code in C++
- And *some* diagrams
  - In OMT, not UML

Pattern Description in GoF Book

- **Pattern Name, Classification, AKA**
- **Intent:** Goals + Reason to Use
- **Motivation:** Example Problem + Context
- **Applicability:** All Suitable Contexts
- **Structure:** Classes & Interactions
- **Participants:** Classes + Roles
- **Collaboration:** Class Interactions
- **Consequences:**
  - Results, Side Effects
  - Tradeoffs
- **Related Patterns**



112

Порождающие паттерны

Известен также под именем

Virtual Constructor (виртуальный конструктор).

Мотивация

Каркасы пользуются абстрактными классами для определения и поддержания отношений между объектами. Кроме того, каркас часто отвечает за создание самих объектов.

Рассмотрим каркас для приложений, способных представлять пользователю сразу несколько документов. Две основных абстракции в таком каркасе – это классы Application и Document. Оба класса абстрактные, поэтому клиенты должны порождать от них подклассы для создания специфичных для приложения реализаций. Например, чтобы создать приложение для рисования, мы определим классы DrawingApplication и DrawingDocument. Класс Application отвечает за управление документами и создает их по мере необходимости, допустим, когда пользователь выбирает из меню пункт **Open** (открыть) или **New** (создать).

Поскольку решение о том, какой подкласс класса Document инстанцировать, зависит от приложения, то Application не может «предсказать», что именно понадобится. Этому классу известно лишь, когда нужно инстанцировать новый документ, а не какой документ создать. Возникает дилемма: каркас должен инстанцировать классы, но «знает» он лишь об абстрактных классах, которые инстанцировать нельзя.

Решение предлагает паттерн фабричный метод. В нем инкапсулируется информация о том, какой подкласс класса Document создать, и это знание выводится за пределы каркаса.

```
Document* doc = CreateDocument();
docs.Add(doc);
doc->Open();
```

```
return new MyDocument
```

Подклассы класса Application переопределяют абстрактную операцию CreateDocument таким образом, чтобы она возвращала подходящий подкласс класса Document. Как только подкласс Application инстанцирован, он может инстанцировать специфические для приложения документы, ничего не зная об их классах. Операцию CreateDocument мы называем *фабричным методом*, поскольку она отвечает за «изготовление» объекта.

Паттерн Factory Method

113

Применимость

Используйте паттерн фабричный метод, когда:

- ❑ классу заранее неизвестно, объекты каких классов ему нужно создавать;
- ❑ класс спроектирован так, чтобы объекты, которые он создает, специфицировались подклассами;
- ❑ класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вы планируете локализовать знание о том, какой класс принимает эти обязанности на себя.

Структура

```
...
product = FactoryMethod()
...
```

```
return new ConcreteProduct
```

Участники

- ❑ **Product** (Document) – продукт:
  – определяет интерфейс объектов, создаваемых фабричным методом;
- ❑ **ConcreteProduct** (MyDocument) – конкретный продукт:
  – реализует интерфейс Product;
- ❑ **Creator** (Application) – создатель:
  – объявляет фабричный метод, возвращающий объект типа Product. Creator может также определять реализацию по умолчанию фабричного метода, который возвращает объект ConcreteProduct;
  – может вызывать фабричный метод для создания объекта Product.
- ❑ **ConcreteCreator** (MyApplication) – конкретный создатель:
  – замещает фабричный метод, возвращающий объект ConcreteProduct.

Отношения

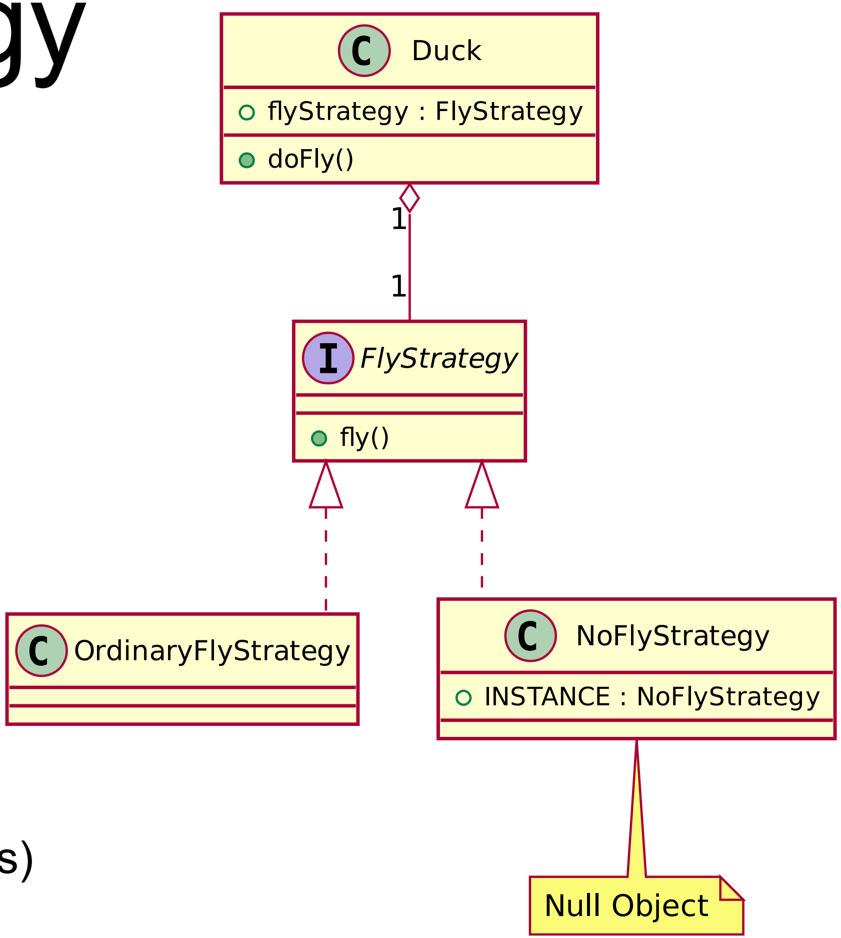
Создатель «полагается» на свои подклассы в определении фабричного метода, который будет возвращать экземпляр подходящего конкретного продукта.

Результаты

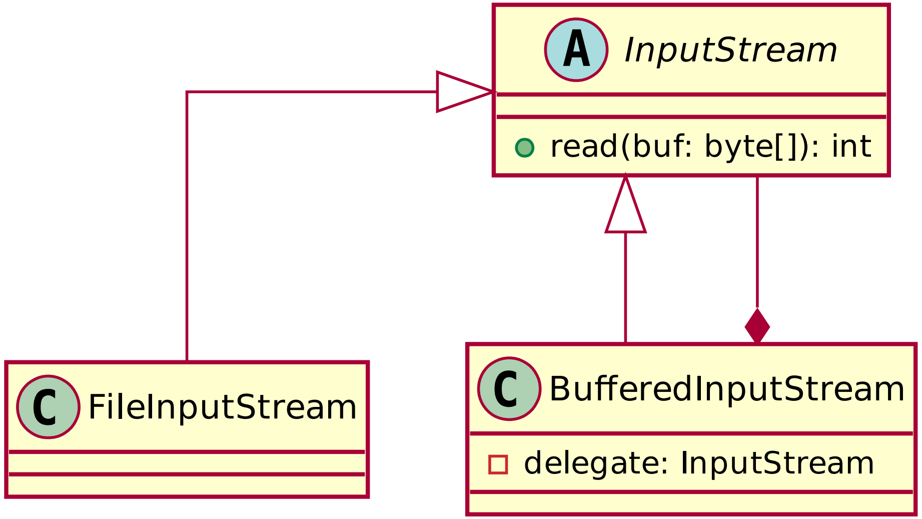
Фабричные методы избавляют проектировщика от необходимости встраивать в код зависящие от приложения классы. Код имеет дело только с интерфейсом класса Product, поэтому он может работать с любыми определенными пользователями классами конкретных продуктов.

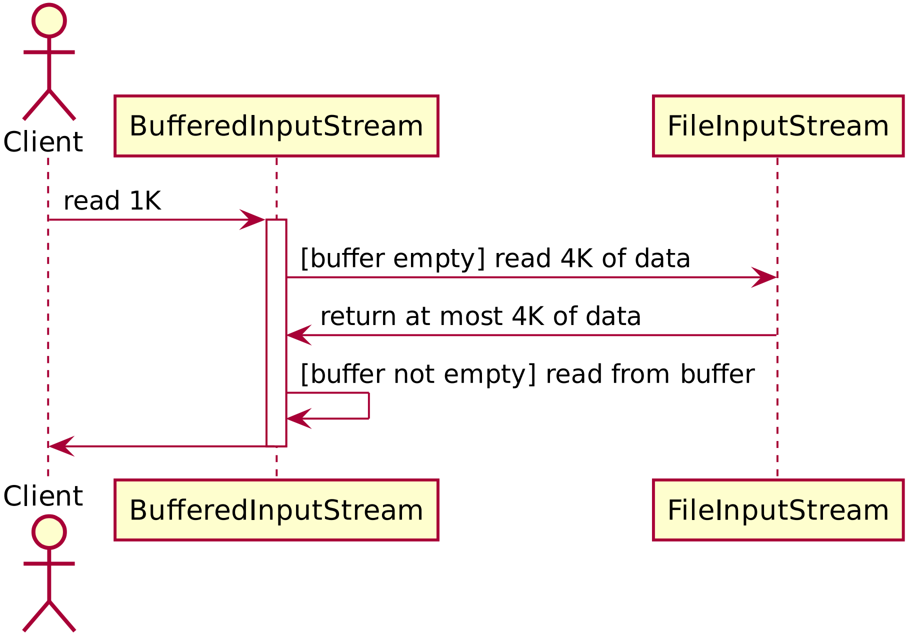
Strategy

- `Duck.`*`fly()`*`. RubberDuck extends Duck.` **Oops!**
- **Solution:**
  - `FlyStrategy.fly()`
  - `Duck <<use>> FlyStrategy.fly()`
  - `RubberDuck` → `(Duck, NoFlyStrategy)`
  - Ordinary `Duck` → `(Duck, OrdinaryFlyStrategy)`
- Strategy offers **Pluggable Behavior**
  - *e.g.* Retry Strategy
- Stategy User is impl-independent
  - Just requires that impl has the specified *interface* (=public methods, pre-and postconditions, invariants)

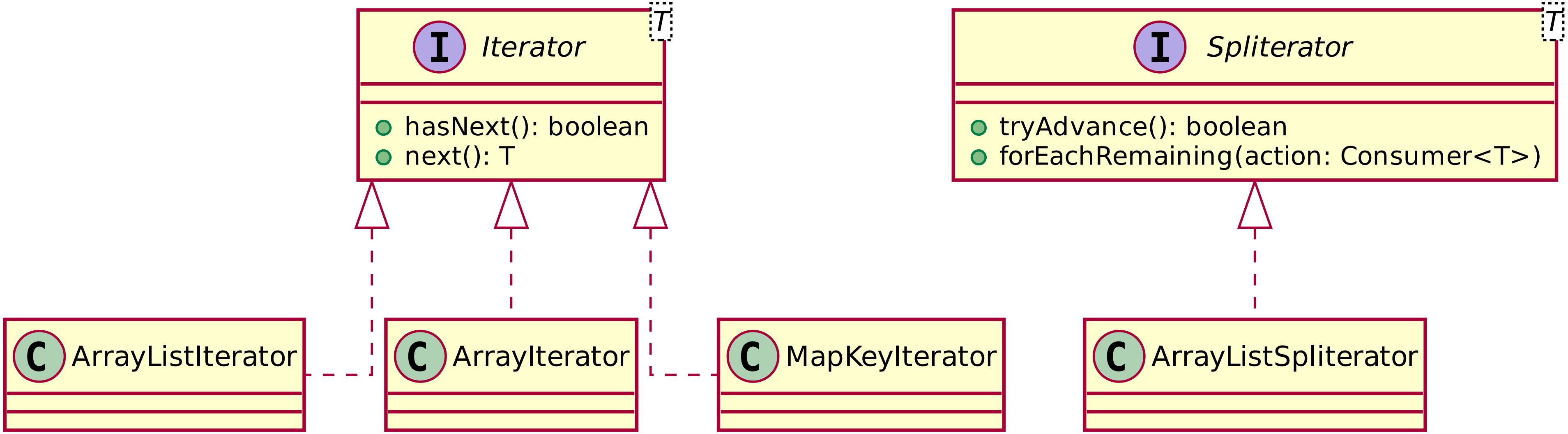


Decorator

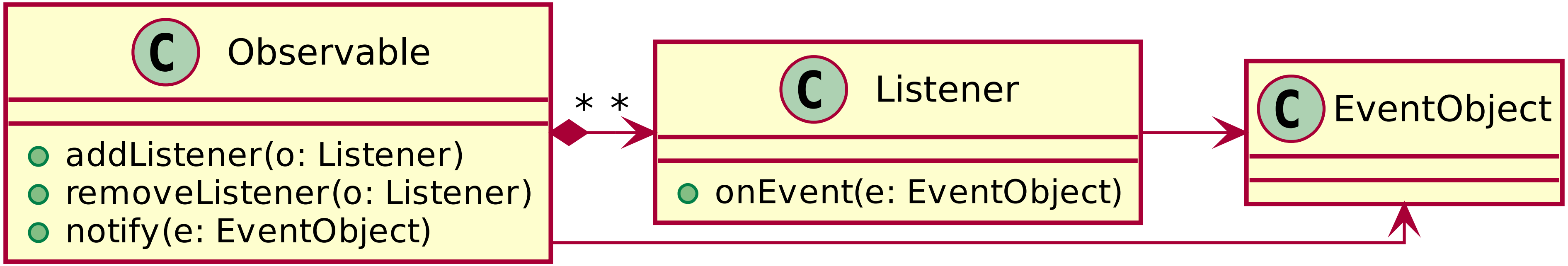

A
InputStream
● read(buf: byte[]): int
C
FileInputStream
C
BufferedInputStream
□ delegate: InputStream


Client
BufferedInputStream
FileInputStream
read 1K
[buffer empty] read 4K of data
return at most 4K of data
[buffer not empty] read from buffer
Client
BufferedInputStream
FileInputStream

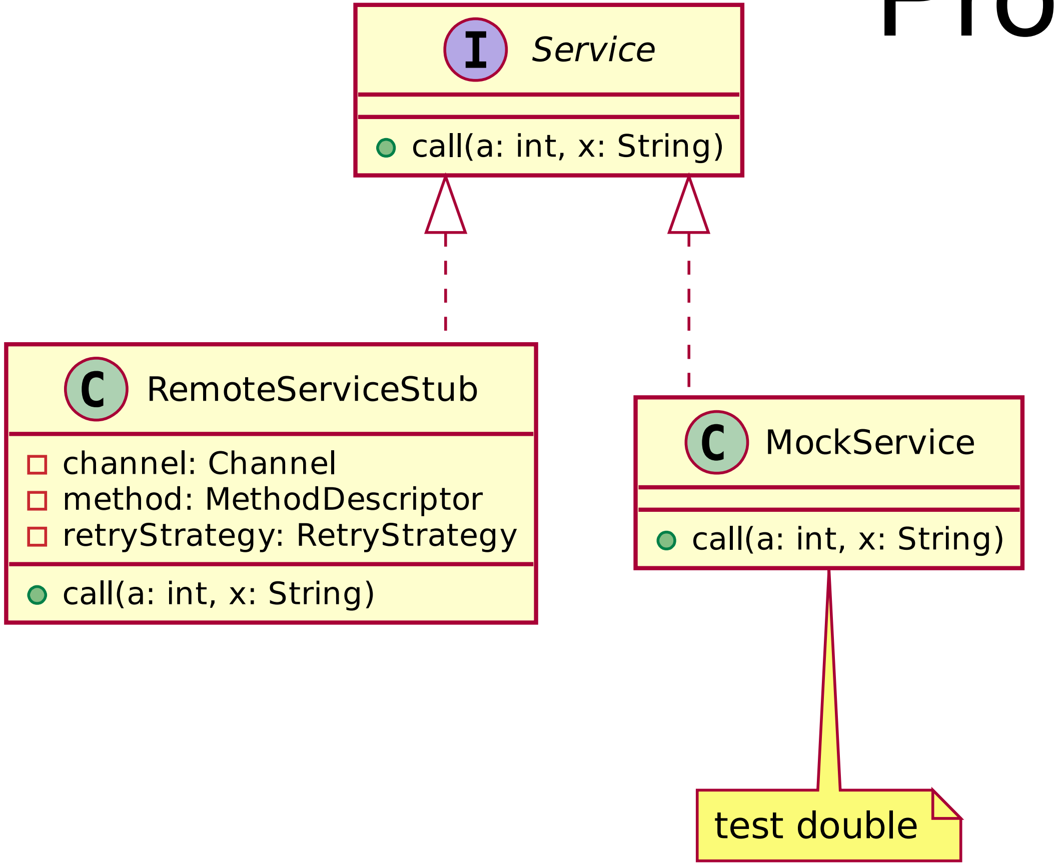
Iterator


I Iterator T
+ hasNext(): boolean
+ next(): T
I Spliterator T
+ tryAdvance(): boolean
+ forEachRemaining(action: Consumer<T>)
C ArrayListIterator
C ArrayIterator
C MapKeyIterator
C ArrayListSpliterator

Observer


C Observable
+ addListener(o: Listener)
+ removeListener(o: Listener)
+ notify(e: EventObject)
* *
C Listener
+ onEvent(e: EventObject)
C EventObject

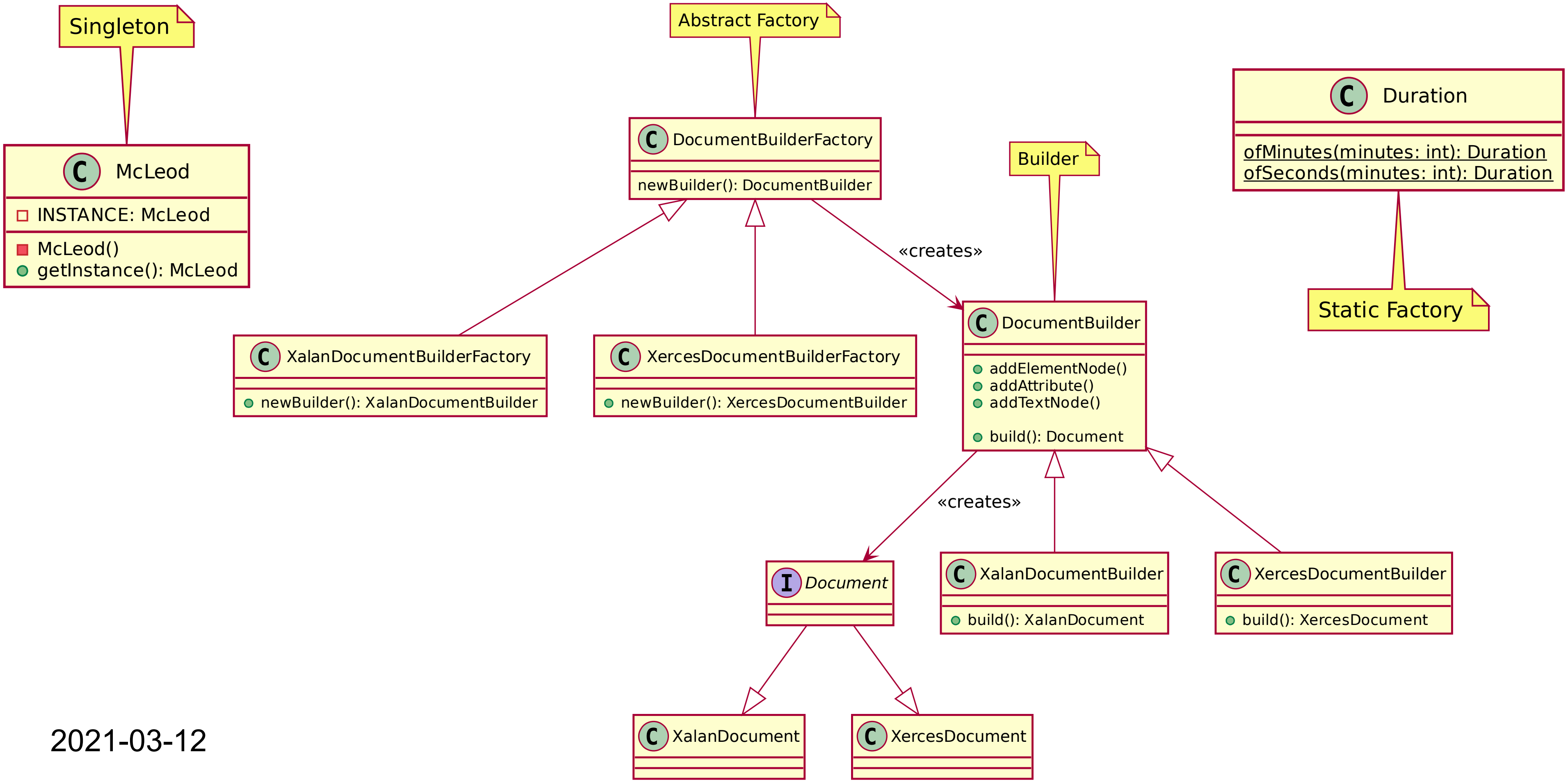
Proxy


I Service
● call(a: int, x: String)
C RemoteServiceStub
□ channel: Channel
□ method: MethodDescriptor
□ retryStrategy: RetryStrategy
● call(a: int, x: String)
C MockService
● call(a: int, x: String)
test double

Proxy vs Decorator:

- Decorators *can* have **similar implementations** as Proxies but serve a **different purpose**
  - *E.g.* protection proxy, logging proxy
- Decorator **adds one or more responsibilities** to an object
- Proxy **controls access** to an object
- Unlike a Decorator:
  - RPC Proxy (*e.g.* RemoteServiceStub) **will not contain a direct reference** to the real subject
  - Virtual Proxy (e.g. reading data from file on demand) **will not contain a direct reference to the real data** until called

Common Creational Patterns



Singleton

McLeod
- INSTANCE: McLeod
- McLeod()
- getInstance(): McLeod

Abstract Factory

DocumentBuilderFactory
newBuilder(): DocumentBuilder

XalanDocumentBuilderFactory
newBuilder(): XalanDocumentBuilder

XercesDocumentBuilderFactory
newBuilder(): XercesDocumentBuilder

«creates»

Builder

DocumentBuilder
addElementNode()
addAttribute()
addTextNode()

build(): Document

«creates»

Document

XalanDocumentBuilder
build(): XalanDocument

XercesDocumentBuilder
build(): XercesDocument

XalanDocument

XercesDocument

Duration
ofMinutes(minutes: int): Duration
ofSeconds(minutes: int): Duration

Static Factory

Some Pattern Pitfalls

- I hate `switch` and `Map`, let's use **polymorphism**
  - https://csis.pace.edu/~bergin/patterns/ppoop.html
- We Need Flexibility Everywhere!
  - ...and then it turns out the interface always has 1 impl
  - System Architecture is what **cannot be changed** without **completely destroying** the System

Recommended Reading (List)

- *Head First Patterns* by Eric & Elizabeth Freeman
  - https://www.ozon.ru/product/head-first-patterny-proektirovaniya-obnovlennoe-yubileynoe-izdanie-frimen-erik-robson-elizabet-211433204
- *Design Patterns: Elements of Reusable Object-Oriented Software* by E. Gamma, R. Helm, R. Johnson, D. Vlissides
  - https://www.ozon.ru/product/priemy-obektno-orientirovannogo-proektirovaniya-patterny-proektirovaniya-135466040/